Lab 4:
Kernel Resource Management

Due date: February 15, 2018, Thursday, in the lab.

Submission deadline: Your solutions will also need to be submitted electronically by 11:59 p.m. of the day when your lab is due. For this lab, it will be due on February 15, 2018, Thursday, 11:59 p.m.

Objectives

In this lab, you will implement three monitors that will be used in our Kernel. These are the **ThreadManager**, the **ProcessManager**, and the **FrameManager**. The code you write will be inspired by the code you have written in Lab 3, in that these three monitors will orchestrate the allocation and freeing of resources.

Files for this Lab

All the files for this lab are available at the following directory on the ECF workstations:

/share/copy/ece353s/lab4

Even though some of the files have the same names, do not reuse files from the previous labs. The same set of files are also available in the Docker image that you can download from the link provided in the course website. Useful commands on using Docker containers have been introduced previously in the Lab 3 handout.

For this lab, you should get the following files:

```
makefile
DISK
Runtime.s
Switch.s
System.h
System.c
List.h
List.c
BitMap.h
BitMap.c
Main.h
Main.c
Kernel.h
Kernel.c
```

The packages called **Thread** and **Synch** in Lab 3 have been merged together into one package, now called **Kernel**. This package contains quite a bit of other material as well, which will be used for later labs. In this and the remaining labs, you will be modifying the **Kernel.c** and **Kernel.h** files. Do not modify the code that is not used in this lab; just leave it in the package.

The **Kernel.c** file contains the following classes and functions:

Thread scheduler functions
Semaphore class
Mutex class
Condition class
Thread class
ThreadManager class
ProcessControlBlock class
ProcessManager class
FrameManager class
AddrSpace class
TimerInterruptHandler
Other interrupt handlers
SyscallTrapHandler
Handler functions

In this lab, you can ignore everything after **TimerInterruptHandler**. The classes called **ThreadManager**, **ProcessManager**, and **FrameManager** are provided in outline, but the bodies of the methods are *not* implemented. You will add implementations to these methods. Some other methods are marked "unimplemented;" those will be implemented in later labs.

The **BitMap** package contains code you will need to read and use; please read and understand the code in this package, but do not modify it.

The **makefile** has been modified to compile the new code. As before, it produces an executable called **os**. Type `make` to compile, and type `make clean` to remove the results of compilation.

You may modify the file **Main.c** while testing, but when you do your final run, please use the **Main.c** file as it was distributed. In the final version of our kernel, the **Main** package will perform all initialization and will create the first thread. The current version performs initialization and then calls some testing functions.

Task 1: Threads and the ThreadManager

In this task, you will modify the **ThreadManager** class and provide implementations for the following methods:

> **Init**
> **GetANewThread**
> **FreeThread**

In our kernel, we will avoid allocating dynamic memory. In other words, we will not use the heap. All important resources will be created at startup time and then we will carefully monitor their allocation and deallocation.

An example of an important resource is **Thread** objects. Since we will not be able to allocate new objects on the heap while the kernel is running, all the **Thread** objects must be created ahead of time. Obviously, we cannot predict how many threads we will need, so we will allocate a fixed number of **Thread** objects (e.g., 10) and re-use them.

When a user process starts up, the kernel will need to obtain a new **Thread** object for it. When a process dies, the **Thread** object must be returned to a pool of free **Thread** objects, so it can be recycled for another process.

Kernel.h contains the line:

```
const MAX_NUMBER_OF_PROCESSES = 10
```

Since each user process in our OS will correspond to one kernel thread (the 1:1 thread model), we will also use this number to determine how many **Thread** objects to place into the free pool initially in the kernel.

To manage the **Thread** objects, we will use the **ThreadManager** class. There will be only one instance of this class, called **threadManager**, and it is created and initialized at startup time in **Main.c**.

Whenever you need a new **Thread** object, you can invoke **threadManger.GetANewThread**. This method should suspend and wait if there are currently none available. Whenever a thread terminates, the scheduler will invoke **FreeThread**. In fact, the **Run** function has been modified in this lab to invoke **FreeThread** when a thread terminates — thereby adding it to the free list — instead of setting its status to **UNUSED**.

Here is the definition of **ThreadManager** as initially distributed:

```
class ThreadManager
  superclass Object
  fields
    threadTable: array [MAX_NUMBER_OF_PROCESSES] of Thread
    freeList: List [Thread]
  methods
    Init ()
    Print ()
    GetANewThread () returns ptr to Thread
    FreeThread (th: ptr to Thread)
endClass
```

When you write the **Init** method, you will need to initialize the array of **Thread**s, and you will need to initialize each **Thread** in the array and set its status to **UNUSED**. (Each Thread will have one of the following as its status: **READY, RUNNING, BLOCKED, JUST_CREATED**, and **UNUSED**.) Threads should have the status **UNUSED** if and only if they are on the **freeList**. You will also need to initialize the **freeList** and place all **Thread**s in the **threadTable** array on the **freeList** during initialization.

You will need to turn the **ThreadManager** into a "monitor." To do this, you might consider adding a **Mutex** lock (perhaps called **threadManagerLock**) and a condition variable (perhaps called **aThreadBecameFree**) to the **ThreadManager** class. The **Init** method will also need to initialize **threadManagerLock** and **aThreadBecameFree**.

The **GetANewThread** and **FreeThread** methods are both "entry methods" in the monitor, so naturally they should obtain the monitor lock in the first statement and release it before they return.

GetANewThread will remove and return a **Thread** from the **freeList**. If the **freeList** is empty, this method will need to wait on the condition of a thread becoming available. The **FreeThread** method will add a **Thread** back to the **freeList** and signal anyone waiting on the condition.

The **GetANewThread** method should also change the **Thread** status to **JUST_CREATED** and the **FreeThread** method should change it back to **UNUSED**.

We have provided code for the **Print** method to print out the entire table of **Thread**s.

Note that the **Print** method disables interrupts. The **Print** method is used only while debugging and will not be called in a running OS so this is okay. Within the **Print** method, we want to get a clean picture of the system state — a "snapshot" — (without worrying about what other threads may be doing), so disabling interrupts seems acceptable. However, the other methods — **Init**, **GetANewThread** and **FreeThread** — must **not** disable interrupts, beyond what is done within the implementations of **Mutex**, **Condition**, etc.

In **Main.c** we have provided a test routine called **RunThreadManagerTests**, which creates 20 threads to simultaneously invoke **GetANewThread** and **FreeThread**. Let us call these the "testing threads" as opposed to the "resource threads," which are the objects that the **ThreadManager** will allocate and monitor. There are 20 testing threads and only 10 resource thread objects.

Every thread that terminates will be added back to the **freeList** (by **Run**, which calls **FreeThread**). Since the testing threads were never obtained by a call to **GetANewThread**, it would be wrong to add them back to the **freeList**. Therefore, each testing thread does not actually terminate. Instead it freezes up by waiting on a semaphore that is never signaled. By the way, the testing threads are allocated on the heap, in violation of the principle that the kernel must never allocate anything on the heap, but this is okay, since this is only debugging code, which will not become a part of the kernel.

In the kernel, we may have threads that are not part of the **threadTable** pool (such as the **IdleThread**), but these threads must never terminate, so there is no possibility that they will be put onto the **freeList**. Thus, the only things on the **freeList** should be **Thread**s from the **threadTable** pool.

You will also notice that the **Thread** class has been changed slightly to add the following fields:

```
class Thread
  ...
  fields
    ...
    isUserThread: bool
    userRegs: array [15] of int     -- Space for r1..r15
    myProcess: ptr to ProcessControlBlock
  methods
    ...
endClass
```

These fields will be used in a later lab. The **Thread** methods are unchanged.

Task 2: Processes and the ProcessManager

In our kernel, each user process will correspond to only one thread. For each process, there will be a single **ProcessContolBlock** object containing the per-process information, such as information about open files and the process's address space. Each **ProcessControlBlock** object will point to a **Thread** object and each **Thread** object will point back to the **ProcessControlBlock**.

There may be other kernel threads that are not associated with any user process. There will only be a small, fixed number of this kind of threads, and they will be created at kernel start-up time.

For now, we will only have a modest number of **ProcessControlBlock**s, which will make our testing job a little easier, but in a real OS this constant would be larger.

```
const MAX_NUMBER_OF_PROCESSES = 10
```

All user processes will be preallocated in an array called **processTable**, which will be managed by the **ProcessManager** object, much like the **Thread** objects are managed by the **ThreadManager** object.

Each user process will be represented by an object of this class:

```
class ProcessControlBlock
  superclass Listable
  fields
    pid: int
    parentsPid: int
    status: int                      -- ACTIVE, ZOMBIE, or FREE
    myThread: ptr to Thread
    exitStatus: int
```

```
    addrSpace: AddrSpace
    fileDescriptor: array [MAX_FILES_PER_PROCESS] of ptr to OpenFile
  methods
    Init ()
    Print ()
    PrintShort ()
endClass
```

Each user process will have a process ID (the field named **pid**). Each process ID will be a unique number, from 1 on up.

Processes will be related to other processes in a hierarchical parent-child tree. Each process will know who its parent process is. The field called **parentsPid** is an integer identifying the parent. One parent may have zero, one, or many child processes. To find the children of process X, we will have to search all processes for processes whose **parentsPid** matches X's **pid**.

The **ProcessControlBlock** objects will be more like C structs than full-blown C++ objects: the fields will be accessed from outside the class but the class will not contain many methods of its own. Other than initializing the object and a couple of print methods, there will be no other methods for **ProcessControlBlock**. We are providing the implementations for the **Init**, **Print** and **PrintShort** methods.

Since we will have only a fixed, small number of **ProcesControlBlock**s, these are resources which must be allocated. This is the purpose of the monitor class called **ProcessManager**.

At start-up time, all **ProcessControlBlock**s are initially FREE. As user processes are created, these objects will be allocated; when the user process dies, the corresponding **ProcessControlBlock** will become FREE once again.

In Unix and in our kernel, the death of a user process is a two stage procedure. First, an ACTIVE process will execute some system call (e.g., **Exit()**) when it wishes to terminate. Although the thread will be terminated, the **ProcessControlBlock** cannot be immediately freed, so the process will then become a ZOMBIE. At some later time, when we are done with the **ProcessControlBlock** it can be freed. Once it is FREE, it is added to the **freeList** and can be reused when a new process is begun.

The **exitStatus** is only valid after a process has terminated (e.g., a call to **Exit()**). So a ZOMBIE process has a terminated thread and a valid **exitStatus**. The ZOMBIE state is necessary just to keep the exit status around. The reason we cannot free the **ProcessControlBlock** is because we need somewhere to store this integer.

For this lab, we will ignore the **exitStatus**. It need not be initialized, since the default initialization (to zero) is fine. Also, we will ignore the ZOMBIE state. Every process will be either ACTIVE or FREE.

Each user process will have a virtual address space and this is described by the field **addrSpace**. The code we have supplied for **ProcessControlBlock.Init** will initialize the **addrSpace**. Although the **addrSpace** will not be used in this lab, it will be discussed later in this document.

The **myThread** field will point to the process's **Thread**, but we will not set it in this lab.

The **fileDescriptors** field describes the files that this process has open. It will not be used in this lab.

Here is the definition of the **ProcessManager** object:

```
class ProcessManager
  superclass Object
  fields
    processTable: array [MAX_NUM_OF_PROCESSES] of ProcessControlBlock
    processManagerLock: Mutex
    aProcessBecameFree: Condition
    freeList: List [ProcessControlBlock]
    aProcessDied: Condition
  methods
    Init ()
    Print ()
    PrintShort ()
    GetANewProcess () returns ptr to ProcessControlBlock
    FreeProcess (p: ptr to ProcessControlBlock)
    TurnIntoZombie (p: ptr to ProcessControlBlock)
    WaitForZombie (proc: ptr to ProcessControlBlock) returns int
endClass
```

There will be only one **ProcessManager** and this instance (initialized at start-up time) will be called **processManager**.

```
processManager = new ProcessManager
processManager.Init ()
```

The **Print()** and **PrintShort()** methods for **ProcessManager** are provided for you. You are to implement the methods **Init**, **GetANewProcess**, and **FreeProcess**. The methods **TurnIntoZombie** and **WaitForZombie** will be implemented in a later lab and can be ignored for now.

The **freeList** is a list of all **ProcessControlBlock**s that are FREE. The status of a **ProcessControlBlock** should be FREE if and only if it is on the **freeList**.

We assume that several threads may more or less simultaneously request a new **ProcessControlBlock** by calling **GetANewProcess**. The **ProcessManager** should be a "monitor," in order to protect the **freeList** from concurrent access. The Mutex called **processManagerLock** is for that purpose. When a **ProcessControlBlock** is added to the **freeList**, the condition **aProcessBecameFree** can be **Signal**ed to wake up any thread waiting for a **ProcessControlBlock**.

Initializing the **ProcessManager** should initialize

the **processTable** array

all the **ProcessControlBlocks** in that array
the **processManagerLock**
the **aProcessBecameFree** and the **aProcessDied** condition variables
the **freeList**

All **ProcessControlBlocks** should be initialized and placed on the **freeList**.

The condition called **aProcessDied** is signaled when a process goes from ACTIVE to ZOMBIE. It will not be used in this lab, but should be initialized nonetheless.

The **GetANewProcess** method is similar to the **GetANewThread** method, except that it must also assign a process ID. In other words, it must set the **pid**. The **ProcessManager** will need to manage a single integer for this purpose. (Perhaps you might call it **nextPid**). Every time a **ProcessControlBlock** is allocated (i.e., every time **GetANewProcess** is called), this integer must be incremented and used to set the process's **pid**. **GetANewProcess** should also set the process's status to ACTIVE.

The **FreeProcess** method must change the process's status to FREE and add it to the free list.

Both **GetANewProcess** and **FreeProcess** are monitor entry methods.

Task 3: The Frame Manager

The lower portion of the physical memory of the BLITZ computer, starting at location zero, will contain the kernel code. It is not clear exactly how big this will be, but we will allocate 1 MB for the kernel code. After that will come a portion of memory (called the "frame region") which will be allocated for various purposes. For example, the disk controller may need a little memory for buffers and each of the user processes will need memory for "virtual pages."

The area of memory called the frame region will be viewed as a sequence of "frames." Each frame will be the same size and we will have a fixed number of frames. For concreteness, here are some constants from **Kernel.h**.

```
PAGE_SIZE = 8192                                -- in hex: 0x00002000
PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME = 1048576 -- in hex: 0x00100000
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512            -- in hex: 0x00000200
```

This results in a frame region of 4 MB, so our kernel would fit into a 5 MB memory.

The frame size and the page size are the same, namely 8K. In later labs, each frame will hold a page of memory. For now, we can think of each frame as a resource that must be managed. We will not really do anything with the frames. This is similar to the dice in the gaming parlor and the forks for the philosophers; we were concerned with allocating them to threads, but did not really use them in any way.

Each frame is a resource, like the dice of the game parlor, or the philosophers' forks. From time to time, a thread will request some frames; the **frameManager** will either be able to satisfy the request, or the requesting thread will have to wait until the request can be satisfied.

For the purposes of testing our code, we will work with a smaller frame region of only a few frames. This will cause more contention for resources and stress our concurrency control a little more. (For later labs, we can restore this constant to the larger value.)

```
  NUMBER_OF_PHYSICAL_PAGE_FRAMES = 27           -- For testing only
```

Here is the definition of the **FrameManager** class:

```
class FrameManager
  superclass Object
  fields
    framesInUse: BitMap
    numberFreeFrames: int
    frameManagerLock: Mutex
    newFramesAvailable: Condition
  methods
    Init ()
    Print ()
    GetAFrame () returns int          -- returns addr of frame
    GetNewFrames (aPageTable: ptr to AddrSpace, numFramesNeeded: int)
    ReturnAllFrames (aPageTable: ptr to AddrSpace)
endClass
```

There will be exactly one **frameManager** object, created at kernel start-up time.

```
frameManager = new FrameManager
frameManager.Init ()
```

With frames (unlike the **ProcessControlBlock**s) there is no object to represent each resource. So to keep track of which frames are free, we will use the **BitMap** package. Please read the code included in this package. The **BitMap** class gives us a way to deal with long strings of bits. We can do things like (1) set a bit, (2) clear a bit, and (3) test a bit. We will use a long bit string to tell which frames are in use and which are free; this is the **framesInUse** field. For each frame, there is a bit. If the bit is 1 (i.e., is "set") then the frame is in use; if the bit is 0 (i.e., is "clear") then the frame is free.

Similar to **threadManager** and **processManager,** the **frameManager** should be organized as a *monitor* class. The **frameManagerLock** is used to make sure only one method at a time is executing in the **FrameManager** code.

We have provided the code for the **Init**, **Print**, and **GetAFrame** methods; you will need to implement **GetNewFrames**, and **ReturnAllFrames**.

The method **GetAFrame** allocates one frame (waiting until at least one is available) and returns the address of the frame. (Since there is never a need to return frames one at a time, there is no "ReturnOneFrame" method.)

When the frames are obtained, the **GetNewFrames** method needs to make a note of which frames have been allocated. It does this by storing the address of each frame it allocates (the address of the first byte in each frame) into an **AddrSpace** object.

An **AddrSpace** object is used to represent a virtual address space and to tell where in physical memory the virtual pages are actually located. For example, for a virtual address space with 10 pages, the **AddrSpace** object will contain an ordered list of 10 physical memory addresses. These are the addresses of the 10 "frames" holding the 10 pages in the virtual address space. However, the **AddrSpace** object contains more information. For each page, it also contains information about whether the page has been modified, whether the page is read-only or writable, etc. The information in an **AddrSpace** object is stored in exactly the format required by the CPU's memory management hardware. In later labs, this will allow us to use the **AddrSpace** object as the current page table for a running user process. At that time, when we switch to a user process, we will have to tell the CPU which **AddrSpace** object to use for its page table. In addition to looking over the code in **AddrSpace**, you may wish to review "The BLITZ Architecture" document, with a focus on page 27 — 31 on page tables and the BLITZ MMU.

The code in method

```
GetNewFrames (aPageTable: ptr to AddrSpace, numFramesNeeded: int)
```

needs to do the following:

(1) Acquire the frame manager lock;
(2) Wait on **newFramesAvailable** until there are enough free frames to satisfy the request;
(3) Do a loop for each of the frames

```
for i = 0 to numFramesNeeded - 1
```

(a) determine which frame is free (find and set a bit in the **framesInUse** BitMap);
(b) figure out the address of the free frame;
(c) execute the following:

```
aPageTable.SetFrameAddr (i, frameAddr)
```

to store the address of the frame which has been allocated;
(d) adjust the number of free frames;
(e) set **aPageTable.numberOfPages** to the number of frames allocated;
(f) unlock the frame manager.

The code in method

```
ReturnAllFrames (aPageTable: ptr to AddrSpace)
```

needs to do more or less the opposite. It can look at **aPageTable.numberOfPages** to see how many are being returned. It can then go through the page table and see which frames it possessed. For each, it can clear the bit.

```
for i = 0 to numFramesReturned-1
  frameAddr = aPageTable.ExtractFrameAddr (i)
  bitNumber = ...frameAddr...
  framesInUse.ClearBit(bitNumber)
endFor
```

It will also need to adjust the number of free frames and "notify" any waiting threads that more frames have become available. To do this, you will need to do a **Broadcast**, because a **Signal** will only wake up one thread. The thread that gets awakened may not have enough free frames to complete, but other waiting threads may be able to proceed.

Also note that there is a possibility of starvation here. It is possible that one large process will be waiting for a lot of frames (e.g., 100 frames). Perhaps there are many small processes which free a few frames here and there, but there are always other small processes that grab those frames. Since there are never more than a few free frames at a time, the big process will get starved.

This particular scenario for starvation, where processes are competing for frames, is a very real danger in an OS and a "real" OS would need to ensure that starvation could not happen. However, in our situation, it is acceptable to provide a solution that risks starvation.

Do not modify the code for the **AddrSpace** class. Do not modify any other files except **Kernel.h** and **Kernel.c.** Do not create global variables (except for testing purposes). Do not modify the methods we have provided.

What to Submit

Complete all three tasks in this handout (or as many tasks as you can), and then submit **Kernel.c** and **Kernel.h** using the command:

```
submitece353s 4 Kernel.c Kernel.h
```

Grading for this Lab

When grading your solution, your TA will ask you to run your solutions in a brief and live demonstration session (including showing the solution code), for each of the three tasks. Task 1 will account for 3 marks, Task 2 will account for 3 marks, and Task 3 will account for 4 marks. Questions may be asked by the TA to check how well you understood your own solutions.